A GLOBAL ROUTING ALGORITHM FOR GENERAL CELLS

Gary W. Clow
Computer Science
California Institute of Technology
Pasadena, California 91125

5127:DF:84

To be published in the Proceedings
of the ACM IEEE 21st Design Automation Conference
Albuquerque, NM, June 25-27, 1984

A Global Routing Algorithm for General Cells

Gary W. Clow

Department of Computer Science
Caltech 256-80
California Institute of Technology
Pasadena, California 91125

Abstract

An algorithm is presented which accomplishes the global routing for a building block or general cell routing problem. A line search technique is employed and therefore no grid is assumed either for the module placements or the pin locations. Instead of breaking the routing surface up into channels, a maze search finds acceptable global routes while avoiding the blocks. Both multi-pin terminals and multi-terminal nets are accomodated.

It is shown that the Lee-Moore grid-based approach is actually a special case of the general search algorithm presented. This algorithm is borrowed from the field of artificial intelligence where it has been applied to many state-space search problems.

Introduction

General cell routing refers to the problem of routing between several blocks of arbitrary size in an integrated circuit layout[1,3,11,12]. This problem arises in several design methodologies. One scenario might be the design of a custom chip in which several individuals participate in the layout. Large components, or macros as they are sometimes called, are produced independently. These components or cells, can then be connected together, along with the pads, to form a complete chip. In another context, some or all of the blocks might be pulled from a design database or cell library. Ultimately, the blocks might be drawn on demand from a paramaterized library by a silicon compiler[4] or perhaps synthesized from a higher level description. Whatever the methodology, ultimately the designer must face the sticky issue of combining the constituent parts of the design into a single chip. It is this step which is both time-consuming and error-prone if done manually.

The goal of a general cell routing system then, is to automate this final step of chip assembly. Classically it has been divided into two phases: cell placement and routing. This paper does not attempt to address the question of floorplanning or automatic cell placement. There are, however, three restrictions placed on the block placement: The blocks must be rectangular, oriented orthogonally, and placed a finite and non-zero distance apart. It is assumed during the global routing phase that an unlimited number of wires may pass between any two cells. With this assumption one is forced either to require the designer to insure sufficient inter-cell spacing in the initial placement or to require the routing system to provide feedback so that the placement can be automatically adjusted. With the latter approach one must be concerned about convergence. Placement adjustment can alter the paths taken during global routing thereby creating inter-cell spacing problems where they did not previously exist. This in turn may lead to another placement adjustment. It has not been shown that this approach is guaranteed to converge even with sufficient restrictions. This is the topic of further research by the author.

Background

The problem of maze searching is closely related to global routing. As early as the 1940's Claude Shannon[13] explored maze searching by building a mechanical mouse which could find its way through a 25-square checkerboard maze. In 1959 Moore[9] presented his now well-known algorithm for maze solving and then in 1961 Lee[8] applied it to wire routing. This grid-expansion algorithm has been modified and adapted and is now commonly known as the Lee-Moore algorithm. In 1969 David Hightower[5] proposed using line segments as the representation instead of

a large grid of points and this greatly improved the efficiency of the algorithm but caused it to fail to find some connections which could be found by a Lee-Moore router. As a result, some routers use Hightower's algorithm for a quick first try, and if it fails, then the full power of the Lee-Moore maze search algorithm is used[6].

This research was motivated by the need to combine the efficiency of the line-segment representation with the thoroughness of the Lee-Moore approach.

State-Space Representation

A state-space metaphor is commonly applied to many real-world problems by artificial intelligence researchers. Much of the early work has concentrated on games such as chess, checkers, and the 15-puzzle[10]. In chess for example, the position of all the pieces at a point in time corresponds to a state in the multi-dimensional space.

In routing, the metaphor is closely tied to the real world. The space is the routing plane and it is, of course, two-dimensional. This makes the state-space representation particularly easy to understand in the case of routing. The state represents the state of the search as it progresses. Let us restrict our attention to routing two-point nets for the moment. A graph is commonly used as an abstraction of the state-space. In the routing plane then, we have a starting point, s, to be connected to a destination point, d. Let s be the *root node* of the graph. As line segments are extended toward the destination, d, each line segment will correspond to an edge in the graph. The first state of the search will be represented by an unconnected graph with two vertices, s and d, and no edges. A path in the routing plane is found when there is a path from s to d in the graph. Intuitively, we will add a new edge each time we extend a line segment toward the target terminal. There will be a vertex along the path every time the route could potentially turn a corner.

In general, we wish to find a *minimal cost path*, where we will assume cost to be the length of the path. So, to each directed edge add a weight corresponding to the distance between the two nodes. Now, the cost of a path is simply the sum of the weights of the edges along the path. Later, we will show how other factors can be considered when calculating the cost of a path, but for now, we will assume we are trying to simply minimize wire length. The term *minimal* is used because there are sometimes several "shortest" rectilinear paths between two points.

The search proceeds by exploring possible paths emanating from s. Were there no obstacles between s and d in the routing plane, then it would be a simple matter to find a Manhattan[8] path between the two points. However, as we know, in the routing domain there will usually be obstacles to be avoided in order to make the connection. To find an acceptable route between two pins then, we must find the least-cost path avoiding all obstacles.

Search Techniques

There are several well-known techniques for searching trees[10,15]. With a slight modification these algorithms may be used for searching graphs.

In our discussion of search algorithms we will need two lists. The first list, OPEN, will contain all those nodes which are on the frontier of the search. They are the only nodes from which the search can expand. Initially, only s is on OPEN. The other list, CLOSED, contains all those nodes which have been encountered during the search and are no longer considered candidates from which the search can be expanded.

A search proceeds by taking a node, n_i, off the OPEN list, generating its successors, adding the successors to the OPEN list, and then placing n_i on the CLOSED list. When searching a graph you must be careful not to have more than one copy of a node *active* at any time. A node is *active* if it is on either the OPEN or CLOSED list.

Generating the successors for node n_i corresponds to finding all the possible points on the routing surface that the search can proceed to from n_i. This is the most difficult step and will be discussed later. For now, we will assume that it is always possible to generate all the successors for a given node once we have pulled that node off the OPEN list.

Search algorithms are often classified by the order in which nodes are placed on, and removed from, the OPEN list. If the order is last-in-first-out the search is called *depth-first*. In depth-first search, as a new node is opened its successors go onto the front of the OPEN list. In depth-first search a depth limit is sometimes used to prevent the algorithm from going too far down the wrong path. If nodes are placed on the OPEN list in first-in-first-out order the search is called *breadth-first*. Depth-first and breadth-first search are examples of *blind* search. They are blind

in the sense that they are not guided by information taken from the problem domain.

Recall that there is a weight associated with each edge in the search graph corresponding to the length of the net segment represented by the edge. We would like for our routing algorithm to find the shortest Manhattan path between the two pins to be connected rather than just any path. We must find then, a way of determining when such a path has been found in order to terminate the search.

In order to discuss the terminating condition we will need to define a function $\hat{g}(n)$. We will denote the cost of a path from s to a node n as $\hat{g}(n)$. Notice that in a tree there is only one path from s to any node n, and so $\hat{g}(n)$is the cost of *the* minimal cost path from s to n.

We can now define a terminating condition for our search algorithms. If we reach a goal node in our search, and it is not possible that any node on OPEN can be on a path of less cost, we may end the search. In order to detect this condition we will need to calculate $\hat{g}(n)$for each node put on OPEN. When all the nodes on OPEN have $\hat{g}(n)$greater than or equal to the $\hat{g}(n)$of a goal node which has been reached, we may stop. We have restricted edge weights to be non-negative by using rectilinear distance as the cost. We are therefore insured that no node on OPEN could be on a path of lesser cost than the path we have found to the goal node since adding non-negative numbers cannot result in a smaller number. If we were to ignore our terminating condition and stop only when no more nodes were left on OPEN, the order in which nodes were placed on OPEN would not matter since all nodes would eventually be expanded. This is called *exhaustive search.*

It is now clear that we may gain by ordering the nodes on OPEN by increasing value of $\hat{g}(n)$. It does not help to expand nodes whose $\hat{g}(n)$is greater than other nodes on OPEN since those with smaller $\hat{g}(n)$values will have to be expanded before the algorithm terminates. At any point now, if a goal node is reached, we also know it is on a minimal cost path because all other nodes on OPEN have $\hat{g}(n)$values at least as great. This algorithm is called *best-first* and is also known as *branch-and-bound.*

The *best-first* algorithm can show a dramatic improvement in time and space efficiency over *blind* searches such as *depth-first* and *breadth-first.* Best-first relies on historical information to predict which nodes are the most likely to be on a *minimal cost path.* The ideal algorithm would operate on *perfect* information thereby always choosing the correct node to expand at each stage of the search. But *perfect* information implies the path is already known, so there is little reason to search. It is possible however, to use *heuristic* (serving to aid discovery) information. This technique is called *heuristic search.* An heuristic tries to predict the candidate nodes which are most likely on a *minimal cost path.*

We shall call an heuristic search algorithm *admissable* if it always finds a *minimal cost path* when a path exists. Nilsson[10] presents an admissable heuristic search algorithm, A*. First, the algorithm will be presented and then, it will be shown how it may be applied to general cell global routing.

Algorithm A*

Let us define an evaluation function, $\hat{f}$, so that its value, $\hat{f}(n)$, at any node n, is an *estimate* of the cost of a minimal cost path *constrained to go through* n. Let the function $k(n_i, n_j)$give the *actual cost* of a minimal cost path between two arbitrary nodes n_i and n_j. Let

$$h(n_i) = \min k(n_i, n_j)$$

where the min is over all paths from n_i to n_j and n_j is the goal node. An *optimal* path from n_i to the goal node is one that achieves $h(n_i)$. Define an *optimal* path from the start node s to some node n by:

$$g(n) = k(s, n)$$

for all n accessible from s.

We can now define a function f, such that, $f(n)$ is the actual cost of an optimal path constrained to go through node n. Let

$$f(n) = g(n) + h(n)$$

We desire our evaluation function, $\hat{f}(n)$, to be an estimate of f. We may take $\hat{f}(n)$to be

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

where $\hat{g}$ is an estimate of g and $\hat{h}$ is an estimate of h.For $\hat{g}(n)$we will use the cost of the path which has been found by the search process in getting to node n.

For $\hat{h}(n)$we will rely on information from the problem domain.

What does $\hat{h}(n)$really represent? It represents our best estimate of the cost of completing the connection between two points using Manhattan geometry while avoiding all obstacles between the two points. Nilsson proves that if $\hat{h}$ is a ***lower bound*** on h, then the A* algorithm is admissible. An obvious choice for $\hat{h}$ then, is the rectilinear (Manhattan) distance from n to the goal. This will always be a lower bound on the actual distance since the route may be forced to go out of its way to avoid obstacles which only increases the length of the wire. In other words, the best you can do using Manhattan geometry is a connection whose length is equal to the rectilinear distance between the two points. Therefore, $\hat{g}$, the rectilinear distance to the goal, will always be a lower bound on g, the actual distance to the goal, and hence algorithm A* will always find an optimal route.

The algorithm proceeds as the other algorithms with s, the start node being placed on OPEN initially. The difference is that nodes are placed on OPEN in ascending order of their $\hat{f}$ values. Nodes are then taken off OPEN and expanded with the most likely candidates, according to the heuristic, expanded first. If a successor is generated which has already been placed on CLOSED, its old and new $\hat{f}$ values must be compared. If its new $\hat{f}$ is less than the old it must be placed back on OPEN. This means a new, shorter path has been found to some intermediate point on the routing surface. The algorithm terminates when the goal node is removed from OPEN to be expanded.

In the implementation it is important to keep pointers from each successor back to its parent node. These pointers provide the means for following back the path to the start node once the search has terminated by finding the goal node. It is also important to note that when a node is moved from CLOSED to OPEN because a shorter path to it has been found, its pointers must be redirected in order to reflect this new shorter path back to the start node.

Generating Successors

The most straightforward way of generating successors is to divide the routing surface up into a grid. The routing surface can then be modelled by setting the grid spacing equal to the minimum wire spacing. Each grid point adjacent to the current node is considered a successor unless the grid point is covered by an obstruction or was the predecessor of the current node. If this model is used with $\hat{g}(n)$ defined to be 0 then it is equivalent to the Lee-Moore[6,8] algorithm.

Using the grid-based approach tends to require large amounts of memory and processor time since so many nodes are expanded. It has the advantage however, of guaranteeing that every possible path is explored.

By exploiting the constraints of the general cell layout, a large improvement can be made over the grid-based approach. First, the only obstacles are rectangular Manhattan cells. It can be observed that optimal paths need only ***hug*** the boundaries of cells if they intervene in the path selection. What is needed then is a method of detecting when a path collides with a cell and a means for generating successors that: (1) extends any path as far toward the goal as is feasible in x and y and (2) ***hugs*** cells (obstacles) as they are encountered. If this technique is applied for generating successors, surprisingly few nodes are generated before an optimal path is found. See figure 1 for an example of the expansion which takes place.

The implementation requires an efficient means of representing the routing surface and its accompanying geometry. The atomic unit of the data structure is the point. Points are linked dynamically to form line segments which can either be edges of boxes (cells) or segments of wire nets. All points are linked to reflect their topological order in both x and y. This maintains their physical relationships. Points are also related logically by the higher level structures to which they belong, namely boxes and wire segments. Therefore, a third set of links is kept to maintain this logical relationship between points.

By maintaining the topological ordering, an efficient means of ***ray-tracing***[14] is used to expand the frontiers of the search.

Extensions

In order for this algorithm to be useful it must handle ***multi-terminal nets*** and ***multi-pin terminals***[12]. Multi-terminal nets are accomodated by approximating a Steiner tree with an adaptation of Dijkstra's[2] minimum spanning tree algorithm. The modification of the spanning tree algorithm considers all line segments in the spanning tree being built as potential connection points. A spanning tree would only consider the pins (vertices) as potential connection points.

Multi-pin terminals are handled by logically

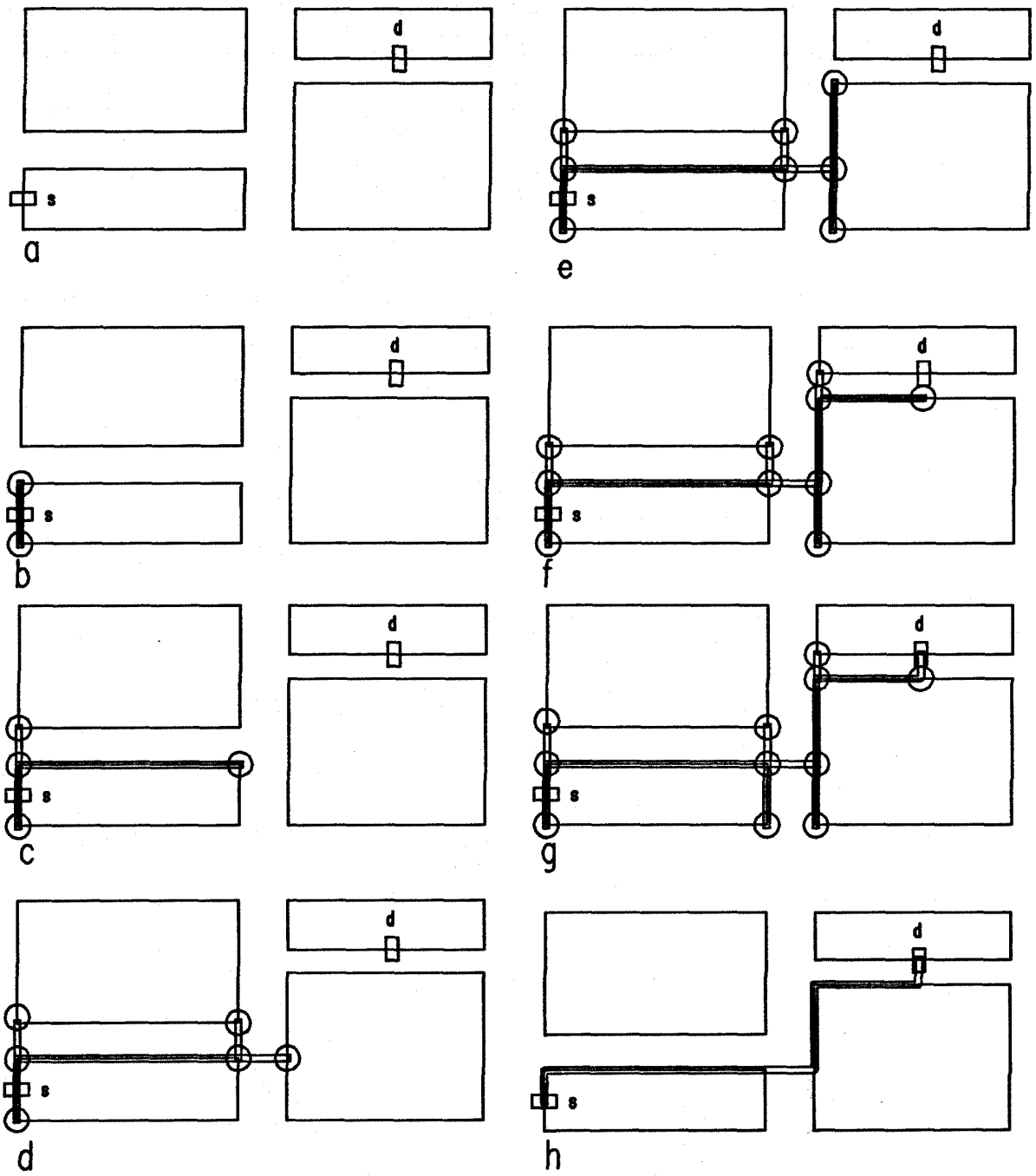

Figure 1. An example of node expansion using A* algorithm.

grouping all pins which belong to a terminal. When a terminal is connected into the tree all the line segments which make up the connecting path as well as all the pins which are associated with the newly connected terminal are brought into the ***connected*** set. This connected set forms the set of potential connections for terminals that are yet to be connected into the tree. The A* algorithm is used to merely find the shortest path which makes each new connection as new terminals are added to the tree.

Because of the generality of the A* algorithm, the heuristic cost function can be used to favor certain classes of routes over others. One problem, that became immediately obvious in the implementation, we will call the ***inverted corner*** (see figure 2). By detecting the inverted corner and penalizing the non-preferred route in the cost function calculation we can cause the router to always take the preferred route. Since both routes have exactly the same length, if a small number, ϵ, is added to the cost of the non-preferred route the algorithm will automatically pick the preferred route.

Another useful extension would be to allow orthogonal ***polygons*** for the cell boundaries. To accomodate the more general cell geometry the procedure which generates successors must be modified so that it leaves no stone unturned. In general, this is equivalent to a grid-less maze search. A form of Sutherland's ray-tracing algorithm[14] must be implemented to accomodate this more general form.

Conclusions

Looking at the classical Lee-Moore approach in the context of the general heuristic search algorithm shows how efficiency improvements can be made for certain restricted problem domains.

The generalized cost function concept also allows other heuristics to be easily implemented. For instance, in the context of a fixed cell placement router (i.e. a router which does not adjust cell placement), a cost function may be associated with what is called ***channel congestion***. Since there are no channels the term is slightly abused, but it refers here to congested passages between adjacent cells. A first-pass route of all nets would reveal congested areas. These congested areas would manifest themselves in the form of several nets hugging the edge of a cell which was ***close*** to an adjacent cell. A second route of the affected nets could penalize those paths which chose the congested area.

Independently routing each net considerably reduces the complexity of the search since the only obstacles are the cells. Classically, nets have been ordered and routed one after another. With this approach nets must avoid other nets as well as cells, greatly increasing the search time. Independent net routing also eliminates the problem of net ordering which can consume a great deal of computing resources in itself.

By routing in the entire routing surface instead of ***channels***, the step of routing surface decomposition is eliminated. The inherent difficulty of the channel decomposition process is compounded by ***pin-splitting***[12,1] and ***inter-channel interference***, and avoiding the whole problem seems to be a good solution.

Actual experience using this algorithm has shown that its efficiency for large problems is very acceptable. The processor time consumed by global routing is always less than the time consumed by detailed routing and layer assignment.

This approach does require a detailed router to follow which does the track assignment. A special algorithm has been developed which dynamically assigns channels based on net interference rather than cell placement. Within the dynamically assigned channel the subnets can be track-assigned using standard channel routing algorithms which try to minimize the number of tracks used. The details of this procedure are beyond the scope of this paper and may be the topic of future publications by this author.

References

[1] A.E. Baratz, ***Algorithms for Integrated Circuit Routing***, PhD. Thesis, Dept. of EE and CS, Massachusetts Institute of Technology (1979).

[2] E.W. Dijkstra, ***A Discipline of Programming***, Prentice-Hall (1976), Ch 22, pp. 154-160.

[3] J.E. Hassett, "Automated Layout in ASHLAR: An Approach to the Problems of 'General Cell' Layout for VLSI," ***Proc. 19th Design Automation Conf.*** (1982), pp. 777-784.

[4] T.S. Hedges, K.S. Slater, G.W. Clow, and T. Whitney, "The Siclops Silicon Compiler," ***ICCC 82 Proceedings***, pp. 277-280.

[5] D. Hightower, "A Solution to Line-Routing Problems on the Continuous Plane," *Proc. 6th Design Automation Workshop*, Miami Beach, Fl., (June 1969), pp. 1-24.

[6] D. Hightower, "The Interconnection Problem: A Tutorial," *Computer*, (April 1974), pp. 18-32.

[7] F.K. Hwang, "The Rectilinear Steiner Problem," *Journal of Design Automation and Fault-Tolerant Computing*, vol. 2, no. 4, (Oct. 1978) pp.303-310.

[8] C. Lee, "An Algorithm, for Path Connections and its Applications." *IRE Trans. On Electronic Computers*, (Sept. 1961), pp. 346-365.

[9] E. Moore, "Shortest Path Through a Maze," *Annals of the Computation Laboratory of Harvard University*, Harvard University Press, Cambridge, Mass., Vol. 30 (1959), pp. 285-292.

[10] N.J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill (1971), Ch 3, pp. 43-78.

[11] B.T. Preas, *Placement and Routing Algorithms for Hierarchical Integrated Circuit Layout*, Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University (1979).

[12] R. Rivest "The 'PI' (Placement and Interconnect) System," *Proc. 19th Design Automation Conference* (1972), pp. 475-481.

[13] C.E. Shannon, "Presentation of the maze-solving machine," *Trans of the 8th Cybernetics Conf.*(1952), Josiah Macy Jr. Foundation, New York, N.Y., pp. 173-180.

[14] I.E. Sutherland, "A Method for Solving Arbitrary-Wall Mazes by Computer," *IEEE Trans. on Computers*, Vol. C-18, No. 12 (Dec. 1969), pp. 1092-1097.

[15] Patrick H. Winston, *Artificial Intelligence*, Addison Wesley,(1984),*Second Edition*,pp. 88-117.

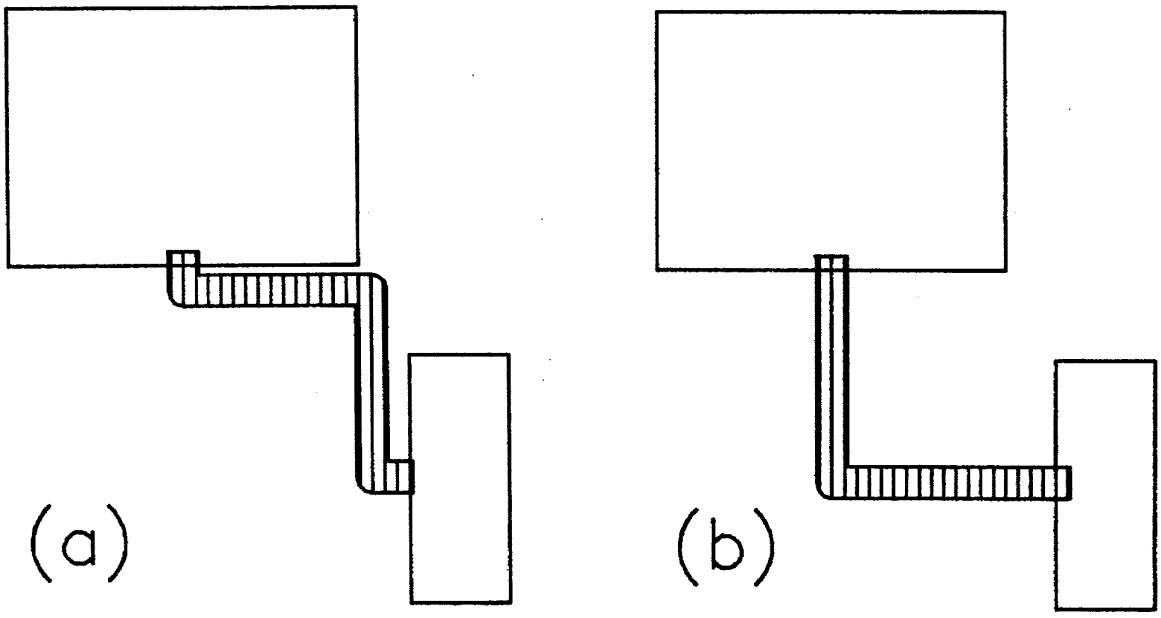

Figure 2. The inverted corner.